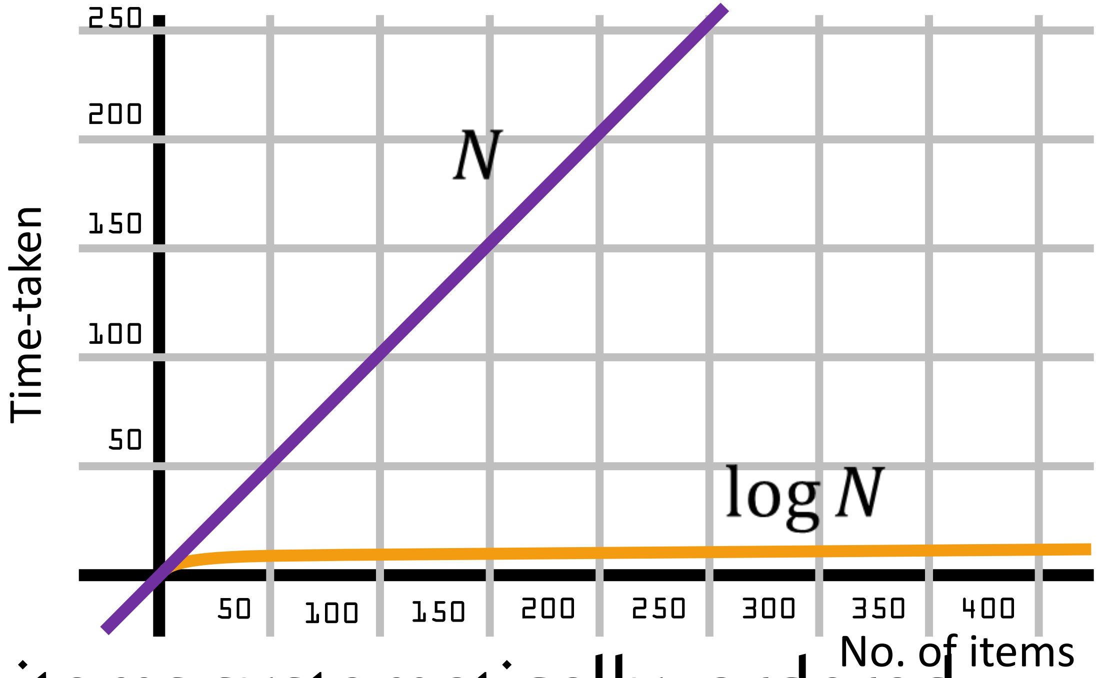
Sorting algorithms

ESC101: Fundamentals of Computing
Nisheeth

What is Sorting?



Useful in itself – internet search and recommendation systems

> Sorting is the process of arranging items systematically, ordered by some criterion

Search within n unsorted elements can take as much as O(n) operations

Makes searching very fast – can search within n sorted elements in just O(log n) operations using Binary Search

Sorting Algorithms

Bubble Sort

Bubble Sort

- Consider an array (5 1 4 2 8). Goal: Sort it in ascending order
- Idea: Repeatedly swap the adjacent elements if they are in wrong order

First Pass

(**5 1** 4 2 8) –> (**1 5** 4 2 8)

(1 **5 4** 2 8) –> (1 **4 5** 2 8),

(1 4 **5 2** 8) –> (1 4 **2 5** 8)

(1 4 2 **5 8**) –> (1 4 2 **5 8**)

Second Pass

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Third Pass

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

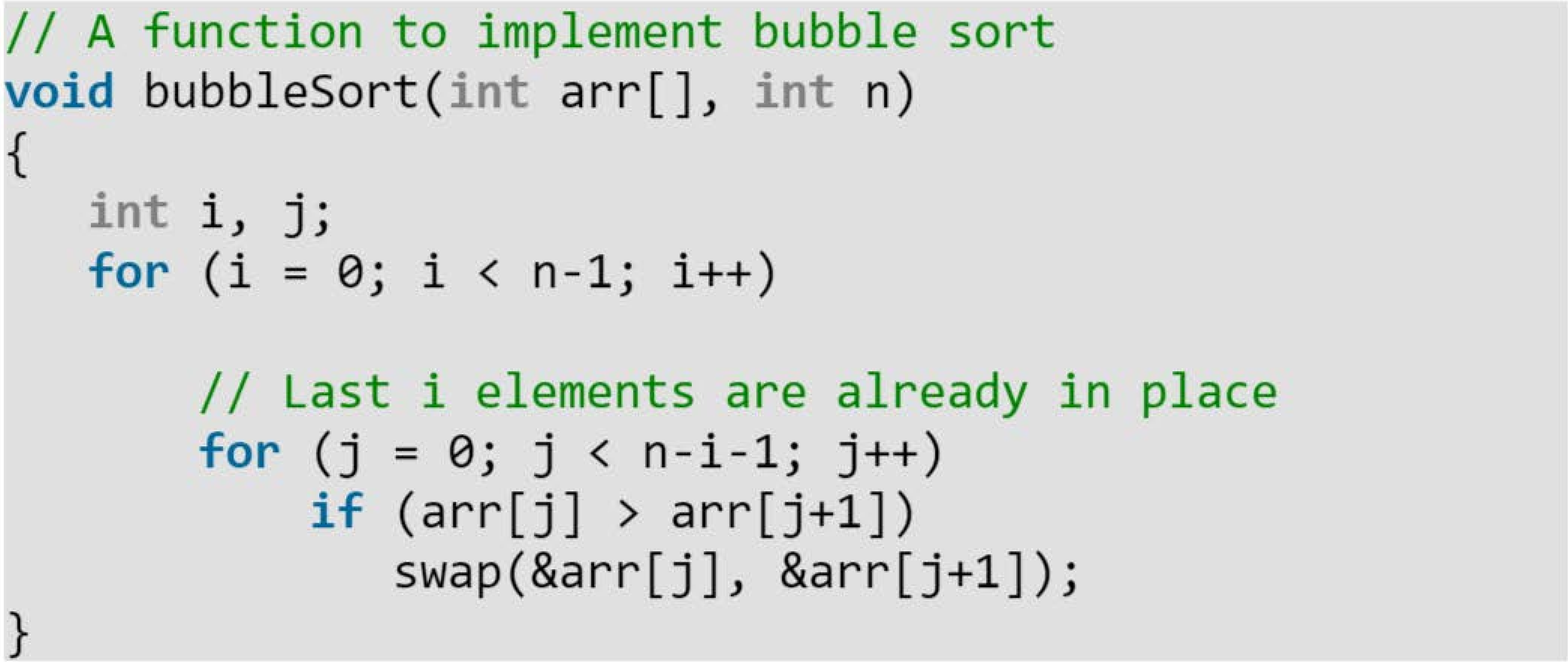
(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

No swaps in this pass, hence done! ☺

Bubble Sort



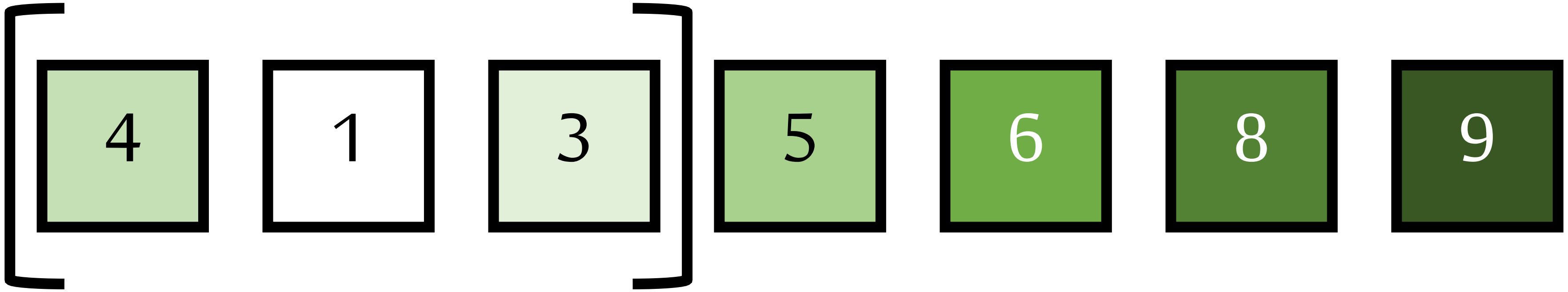
```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

Sorting Algorithms

Selection Sort

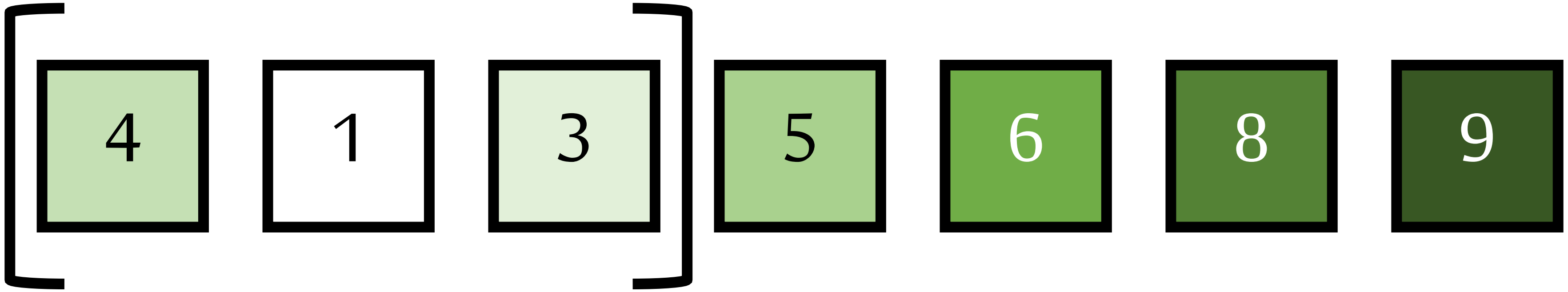
Selection Sort

- Another very simple sorting algorithm
- Like binary search, maintains *active range* $a[0:R]$ with $0 \leq R < N$
  - Initially the active range is entire array i.e. $R = N - 1$
- We will ensure two things
  - At all points of time, the non-active portion will be sorted in ascending order i.e. for all $R \leq i < j$ we will ensure $a[i] \leq a[j]$
  - The non-active elements will never be smaller than the elements in the active range i.e. if $i \leq R < j$ then $a[i] \leq a[j]$
- Active region will shrink by one element at each step (details in next class)



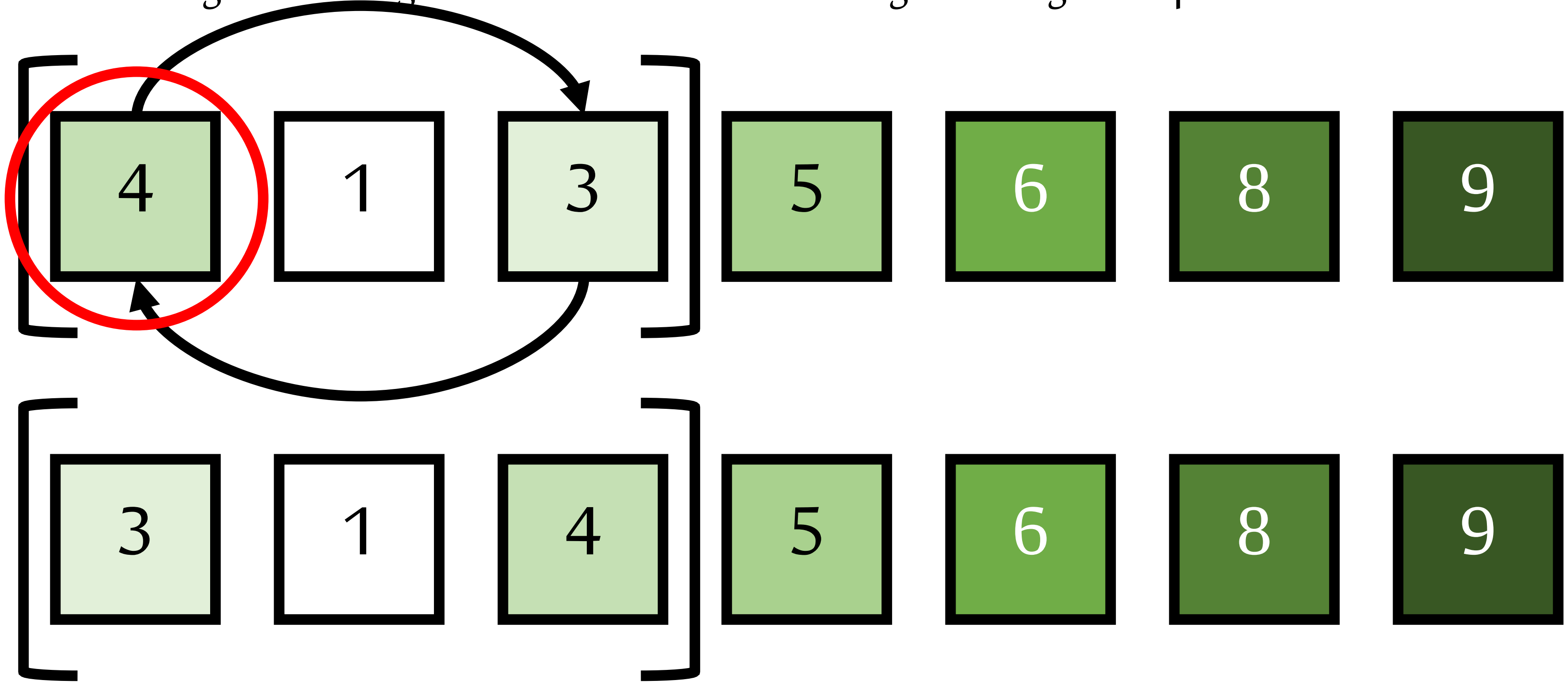
Selection Sort

- Already saw Bubble Sort. Selection sort is another very simple sorting algo
- Like binary search, maintains *active range* $a[0:R]$ with $0 \leq R < N$
  - Initially the active range is entire array i.e. $R = N - 1$
- We will ensure two things
  - At all points of time, the non-active portion will be sorted in ascending order i.e. for all $R \leq i < j$ we will ensure $a[i] \leq a[j]$
  - The non-active elements will never be smaller than the elements in the active range i.e. if $i \leq R < j$ then $a[i] \leq a[j]$
- The active region will shrink by one element at each step



Selection Sort

- Once an element goes to non-active region, we never touch it again ☺
- To maintain our conditions and still shrink the active region
  - We find the largest element in the active region
  - Bring it to the right-most end of the active region using a swap



Verify that our conditions still hold

Selection Sort

Exercise: write a recursive version

xercise: convert this to proper C code

SELECTION SORT

```
1. Given: Array $a$ with $N$ elements
2. For $R = N - 1; R > 0; R --$          // Initial active range is full array
    1. $i \leftarrow \mathrm{FINDMAX}(a, 0, R)$          // Location of largest element in $a[0, R]$
    2. $\mathrm{SWAP}(a, i, R)$          // Bring largest element to the end
```

SWAP

```
1. Given: Array $a$, location $i, j$
2. Let $tmp \leftarrow a[i]$
3. Let $a[i] \leftarrow a[j]$
4. Let $a[j] \leftarrow tmp$
```

FINDMAX

```
1. Given: Array $a$, locations $i, j$
2. Let $k \leftarrow i, \max = a[k]$
3. For $l = i; i \leq j; l++$
    1. If $a[l] > \max, \max = a[l], k = l$
4. Return $k$
```

Time Complexity

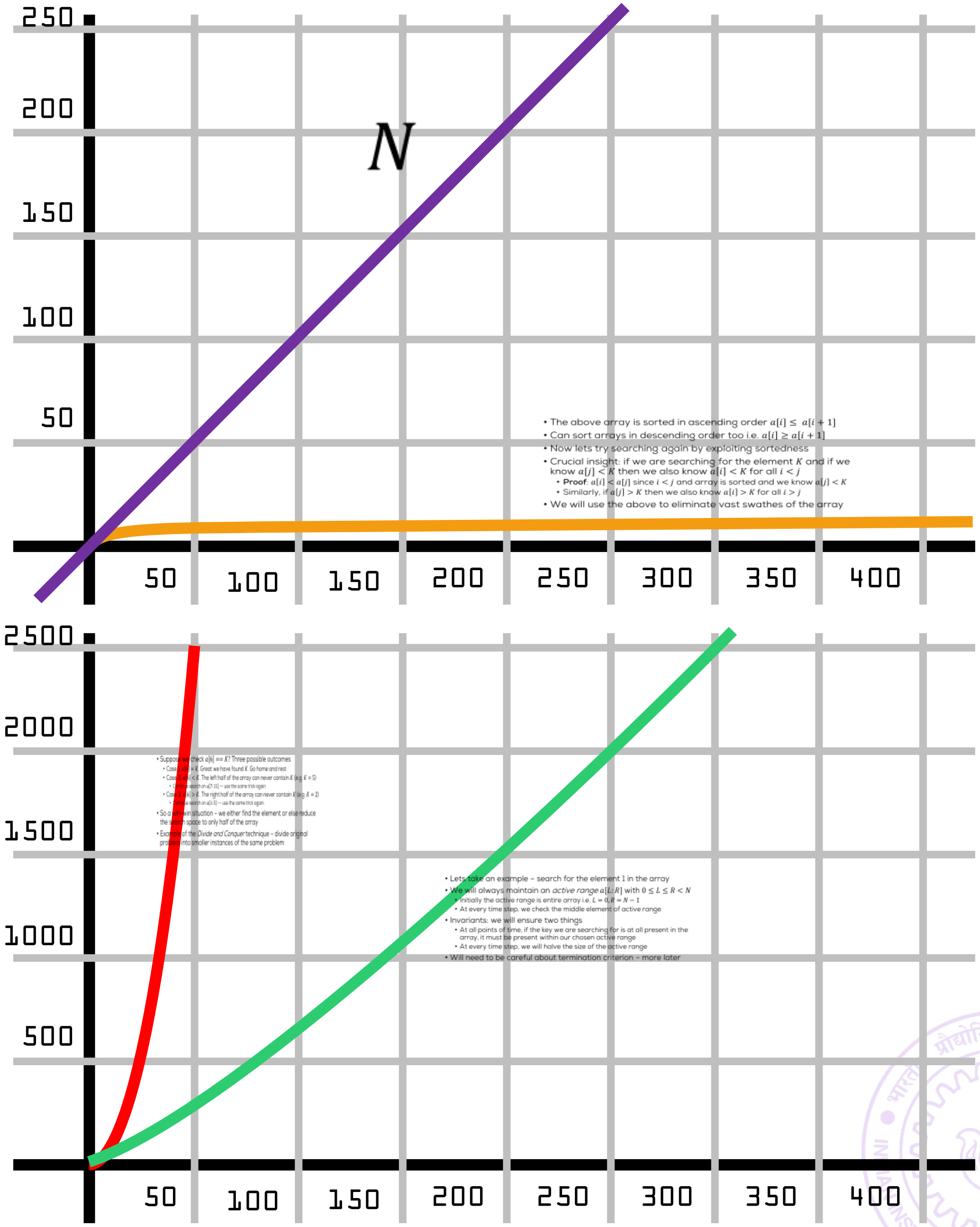
- Let $T(N)$ be the time taken for selection sort to sort N elements
- Let $M(N)$ be the time taken to find location of max of N elements
- At any time step when active region is $[0\colon R]$, we do two things
  - Find the largest element within the active region – takes time $M(R+1)$
  - Swap the largest element with the element at $a[R]$ - takes time c (const)
- Thus, we have $T(N) \leq M(N) + c + T(N-1)$
- It is easy to show that $M(N) \leq d \cdot N$ for all N for some constant d
- Exercise: expand the recurrence as before and show that

  $$T(N) \leq \mathcal{O}(N^2)$$

  Assume $T(1) \leq c$
- Notice that selection sort (also bubble sort) doesn't need any extra memory (except a few tmp variables to store one integer each) – *in-place sorting*

Summary so far..

- Applications of sorting: ranking, recommendation, internet search
- Brute force search $\mathcal{O}(N)$
- Fast searches on sorted arrays: binary search $\mathcal{O}(\log N)$
- Bubble sort $\mathcal{O}(N^2)$
- Selection sort $\mathcal{O}(N^2)$
- Next: fast sorting $\mathcal{O}(N \log N)$
    - Merge Sort
    - Quick Sort



- The above array is sorted in ascending order $a[i] \leq a[i+1]$
- Can sort arrays in descending order too i.e. $a[i] \geq a[i+1]$
- Now lets try searching again by exploiting sortedness
- Crucial insight: if we are searching for the element K and if we know $a[j] < K$ then we also know $a[i] < K$ for all $i < j$
    - **Proof**: $a[i] < a[j]$ since $i < j$ and array is sorted and we know $a[j] < K$
    - Similarly, if $a[j] > K$ then we also know $a[i] > K$ for all $i > j$
- We will use the above to eliminate vast swathes of the array

- Suppose we check $a[6] == K$? Three possible outcomes
    - Case 1: $a[6] = K$. Great we have found K. Go home and rest
    - Case 2: $a[6] < K$. The left half of the array can never contain K (e.g. $K = 5$)
        - Continue search on $a[7:11]$ -- use the same trick again
    - Case 3: $a[6] > K$. The right half of the array can never contain K (e.g. $K = 2$)
        - Continue search on $a[1:5]$ -- use the same trick again
- So a win-win situation – we either find the element or else reduce the search space to only half of the array
- Example of the *Divide and Conquer* technique – divide original problem into smaller instances of the same problem

- Lets take an example – search for the element 1 in the array
- We will always maintain an *active range* $a[L:R]$ with $0 \leq L \leq R < N$
    - Initially the active range is entire array i.e. $L = 0, R = N - 1$
    - At every time step, we check the middle element of active range
- Invariants: we will ensure two things
    - At all points of time, if the key we are searching for is at all present in the array, it must be present within our chosen active range
    - At every time step, we will halve the size of the active range
- Will need to be careful about termination criterion – more later

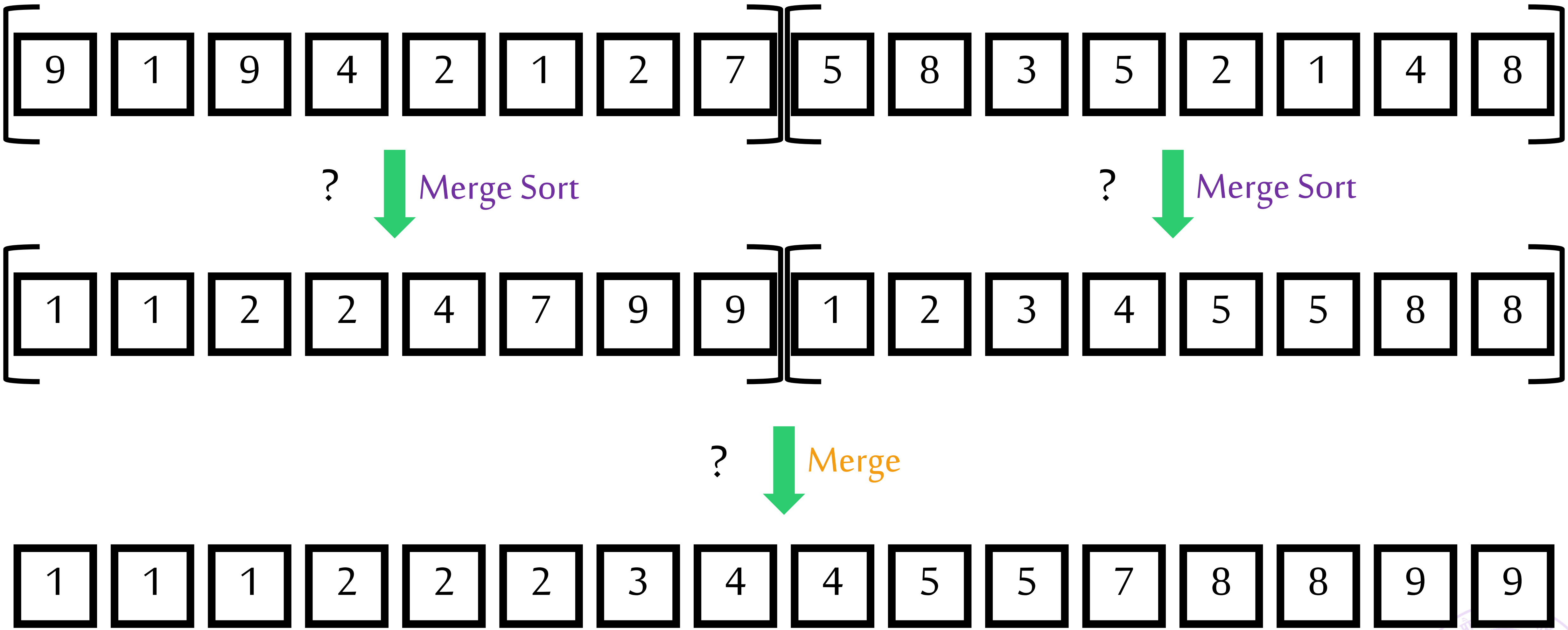
Partition based Sorting Techniques

- Let $T(N)$ be the time taken for selection sort to sort N elements
- Let $M(N)$ be the time taken to find location of max of N elements
- For selection sort, we saw that $T(N) \leq M(N) + c + T(N-1)$
- Active region shrank too slowly which gave us $T(N) \leq \mathcal{O}(N^2)$
- Selection sort (also bubble sort) is quite expensive (imagine $\mathcal{O}(N^2)$ time complexity for $N = 1{,}000{,}000$ items ☹) – much better can be done
- Will study two sorting algorithms based on divide and conquer technique
- Both algorithms will split an array of N elements into two arrays, sort each smaller array and then do some clean up operations
    - Merge Sort: popular for sorting large scale databases
    - Quick Sort: extremely popular in general (see qsort() in stdlib.h)

Sorting Algorithms

Merge Sort

Merge Sort



[9 1 9 4 2 1 2 7] [5 8 3 5 2 1 4 8]

? Merge Sort ? Merge Sort

[1 1 2 2 4 7 9 9] [1 2 3 4 5 5 8 8]

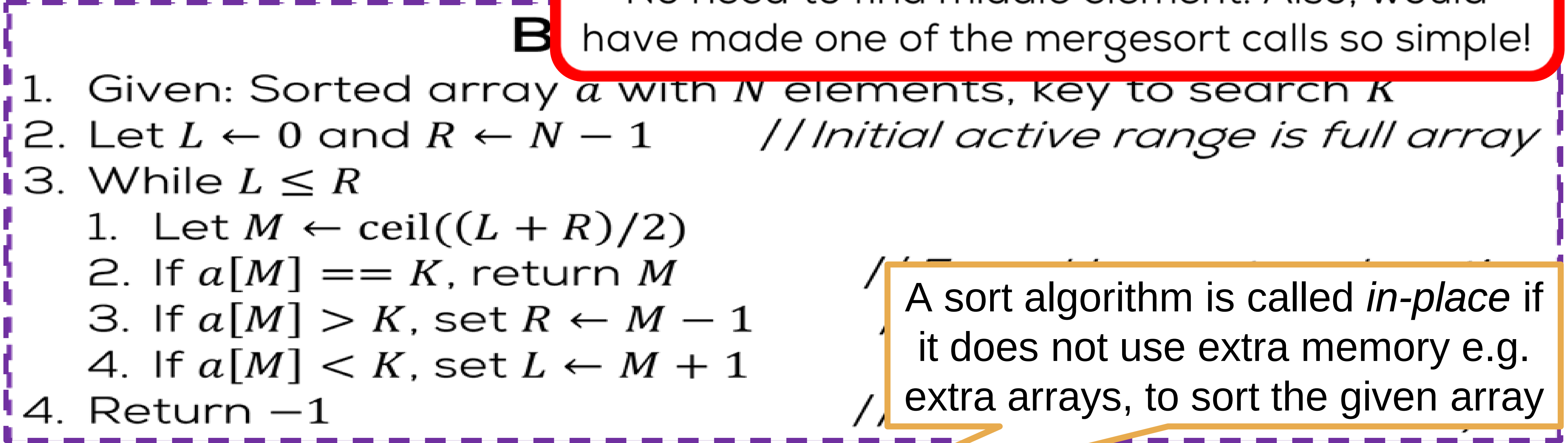
? Merge

1 1 1 2 2 2 3 4 4 5 5 7 8 8 9 9

Trick: Merging two sorted arrays is very easy!

Merge Sort

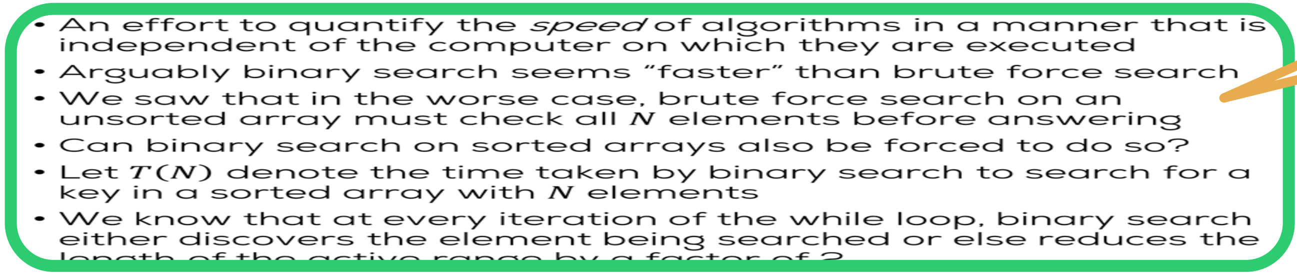
Why didn't we split as $[0\colon N-2], [N-1\colon N-1]$? No need to find middle element. Also, would have made one of the mergesort calls so simple!



B

```
1. Given: Sorted array $a$ with $N$ elements, key to search $K$
2. Let $L \leftarrow 0$ and $R \leftarrow N-1$        // Initial active range is full array
3. While $L \leq R$
    1. Let $M \leftarrow \text{ceil}((L+R)/2)$
    2. If $a[M] == K$, return $M$
    3. If $a[M] > K$, set $R \leftarrow M-1$
    4. If $a[M] < K$, set $L \leftarrow M+1$
4. Return $-1$
```

A sort algorithm is called *in-place* if it does not use extra memory e.g. extra arrays, to sort the given array



- An effort to quantify the *speed* of algorithms in a manner that is independent of the computer on which they are executed
- Arguably binary search seems "faster" than brute force search
- We saw that in the worse case, brute force search on an unsorted array must check all N elements before answering
- Can binary search on sorted arrays also be forced to do so?
- Let $T(N)$ denote the time taken by binary search to search for a key in a sorted array with N elements
- We know that at every iteration of the while loop, binary search either discovers the element being searched or else reduces the length of the active range by a factor of 2

Time Complexity

If we had split as $[0:N-2], [N-1:N-1]$ then $T(N) \leq T(N-1) + T(1) + M(N)$ would have given us $T(N) = \mathcal{O}(N^2)$ (divide properly to rule powerfully ☺)

- Let $T(N)$ be the time take sort N elements
- Let $M(N)$ be time merging two sorted arrays with total N elements
- Thus, we have $T(N) \leq 2 \cdot T(N/2) + M(N) + d$ (d: time to find middle index)
- We will show next that we can do $M(N) \leq c \cdot N$ time
- This recurrence is a bit harder to solve but we can still try

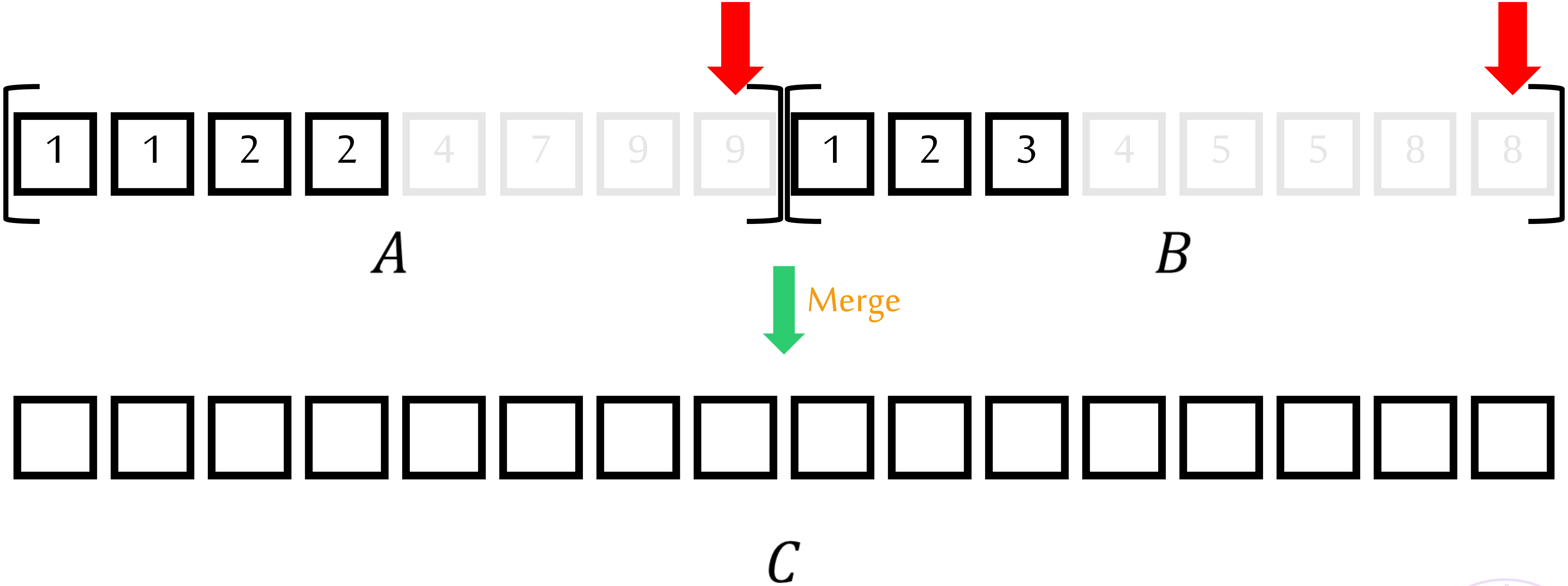
$$T(N/2) \leq 2 \cdot T(N/4) + c \cdot N/2 + d$$
$$T(N) \leq 4 \cdot T(N/4) + 2c \cdot N + (1+2) \cdot d$$
$$T(N) \leq 2^k \cdot T\left(N/2^k\right) + kc \cdot N + 2^k \cdot d$$

- Set $k = \text{ceil}(\log N)$ and use $T(1) \leq c$ to get $T(N) \leq \mathcal{O}(N \log N)$
- The version of merging we will show uses extra $\mathcal{O}(N)$ memory. Can you develop a version that uses only 2-3 extra integer variables i.e. an *in-place* version of merge sort?

The Merge Operation

- Given 2 arrays int $a[M], b[N]$; both sorted in ascending order
- Want a combined array int $c[M+N]$; sorted in ascending order
- Will maintain *active ranges* for both arrays $a[0{:}R_1]$ and $b[0{:}R_2]$ with $0 \leq R_1 < M$ and $0 \leq R_2 < N$
  - Initially the active ranges are the entire arrays i.e. $R_1 = M - 1, R_2 = N - 1$
- At all points of time, we will ensure that elements in the non-active regions of the arrays would have been inserted into c at their proper locations
- At least one active region will shrink by one element at each step
- Trick: the largest element of c can be found in $\mathcal{O}(1)$ time since the arrays a, b are sorted. If unsorted it would have taken $\mathcal{O}(M+N)$

The Merge Operation



9 is larger: B wins again

The Merge Operation

- Given a (non-sorted) array `int a[N];` count the number of swaps A swap is a pair $0 \leq i \neq j < N$ such that $i < j$ but `a[i] > a[j]`
  - This problem is related to a ranking metric known as *area under the ROC curve*. Check it out if interested
- We have two arrays of N numbers `int P[N],F[N];` containing 12th marks of N students each who cleared and did not clear JEE
  - Find out the number of students who did not clear JEE but had 12th std marks more than at least 50% of students who did clear JEE
- Solve these problems faster than $O(N^2)$ time (Hint may involve sorting). Assume you have a routine that can sort N elements in $O(N \log N)$ time – will see such methods soon.

Sorting Algorithms

Quick Sort

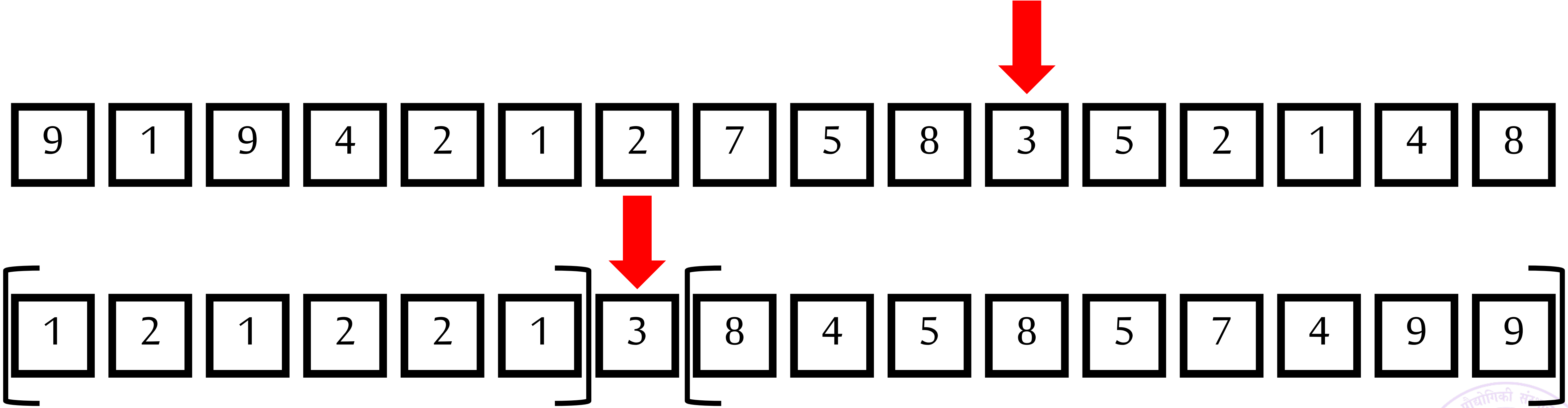
Quick Sort

- Very popular sorting algorithm – try this before anything else
- $\mathcal{O}(N \log N)$ time complexity but in practice faster than merge sort
- Like selection sort, merge sort lazily divides the array into two equal halves, sorts the halves recursively and then spends time merging them
- Quick sort is more careful in splitting the array so that no need for merging once the subarrays are sorted!
- Based on a cool trick known as *partitioning*
- Analysis of quick sort is much more advanced – in worst case quicksort takes $\mathcal{O}(N^2)$ time but this happens very very rarely.
- On average quicksort enjoys $\mathcal{O}(N \log N)$ time complexity

The Partition Technique

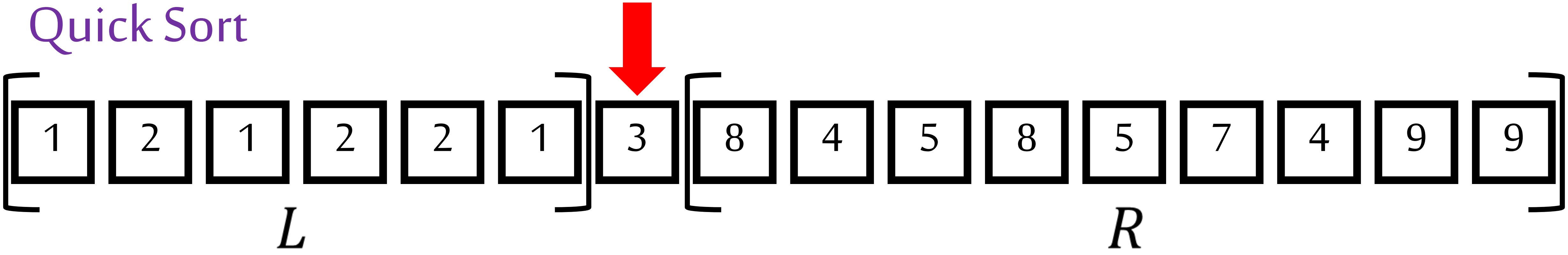
- Given array int $a[N]$ and any element of the array p (called pivot)
- Create a new array int $b[N]$ which is arranged as follows

$$[\text{elements of } a \le p, \qquad p, \qquad \text{elements of } a \ge p]$$



- Notice that left and right halves are not sorted yet! 😐
- Also, the two halves are not balanced (of same size) either 😐

Quick Sort



1	2	1	2	2	1	3	8	4	5	8	5	7	4	9	9

- Notice that even though the subarrays L, R not sorted, every element of L is smaller than or equal to every element of R
- This means that if we sort L, R recursively, no need to merge ☺
- Key to quicksort's success – partition and recursively sort!
- Will discuss a partition algorithm that ensures a stricter condition

  [elements of $a < p$, all instances of p, elements of $a > p$]
- However, our algorithm will use extra memory
- Time complexity analysis of quicksort beyond scope of ESC101

Quick Sort

QUICKSORT

```
1. Given: Array $a$ wit
2. If $N < 2$ return $a$                          or singleton array is sorted
3. Let $p \leftarrow$ CHOOSEPIVOT($a$)                   //Choose a pivot value
4. Let $(b, i) \leftarrow$ PARTITION($a, p$)       //Partition along chosen pivot
5. QUICKSORT($b[0 : i - 1]$)                          //Sort the left half
6. QUICKSORT($b[i + 1, N - 1]$)                      //Sort the right half
7. Return $b$
```

i is the new location of the pivot element

Common choices for pivot value

- $a[0]$ or $a[N-1]$ i.e. end elements
- $a[i]$ for $i \sim \mathrm{random}(N)$ i.e. a random element
- MEDIAN(a) i.e. median element of the array

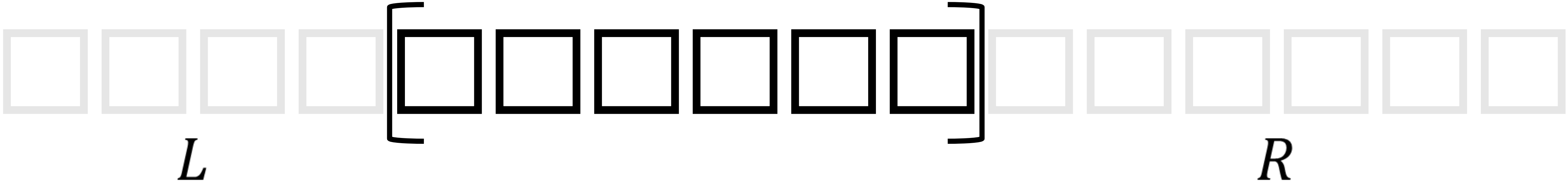
Most popular, inexpensive

Also common, inexpensive

Ensures balanced partition but expensive

The Partition Procedure

- The partition procedure maintains an interesting structure of one active region sandwiched between two inactive regions ☺



- Elements in the left inactive region are strictly less than the pivot, those in right invariant region strictly larger than pivot
- What about element(s) equal to the pivot – need to be careful
- We will see a visualization of the partition procedure in action
- Note: these regions will be maintained on a separate array and not the original array – we will only take a simple left-to-right pass on the original array

We are sure now that any blank spaces left must be occurrences of pivot 4 that we omitted earlier

ll occurrences of the pivot element

e done with non-pivot elements

PIVOT = 4

9	1	9	4	2	1	2	7	5	8	3	5	2	1	4	8

L R

$8 > 4$ i.e. belongs to R

Can’t insert 4 now as there are still elements of L/R left to be processed. If we insert 4 now, we may violate our conditions later

The Partition Procedure

PARTITION

```
1. Given: Array $a$ with $N$ elements, pivo
2. Let int $b[N], L \leftarrow 0, R \leftarrow N$          // Initi
3. For $i = 0; i < N; i{+}{+}$
    1. If $a[i] < p$, let $b[L] \leftarrow a[i]$, and $L{+}{+}$     // We found a left element
    2. If $a[i] > p$, let $b[R] \leftarrow a[i]$, and $R{-}{-}$ // We found a right element
4. For $i = L; i \le R; i{+}{+}$
    1. Let $b[j] \leftarrow p$          // Fill up th                t
5. Return $(b, R)$
```

Verify that after the first loop has ended, we must have $L < R$ i.e. some space left for pivot

In fact, the entire range $b[L{:}R]$ is filled with the pivot element

R has to be (one of) the new location(s) of the pivot element

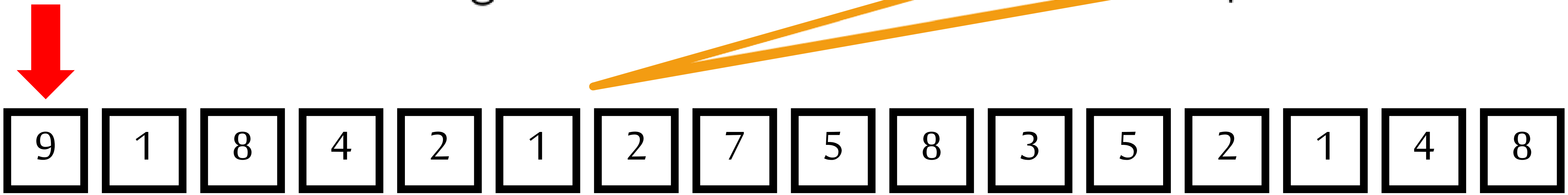
Hint: the in-place algorithm uses an identical notion of inactive regions but swaps elements at the boundaries of the regions which are wrongly placed

Explore/invent yourself an *in-place* partitioning algorithm

Choice of Pivot

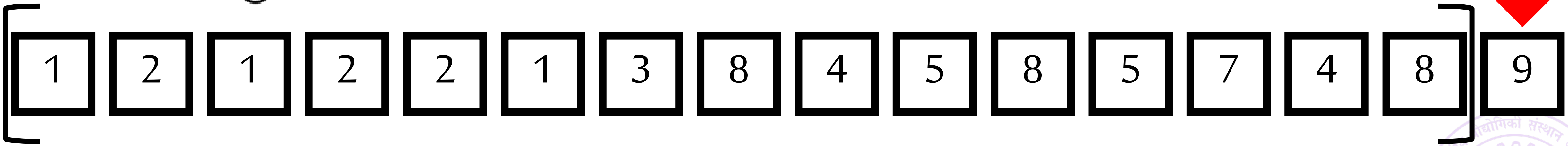
- Most crucial step in quicksort – m
- Suppose we are so unlucky that t smallest or the largest element o

Ironically, if the array is already sorted and we use end elements as pivots, then quicksort takes $\mathcal{O}(N^2)$ time ☹



9	1	8	4	2	1	2	7	5	8	3	5	2	1	4	8

- Choosing an element close to the median is most beneficial



1	2	1	2	2	1	3	8	4	5	8	5	7	4	8	9

- Quicksort becomes selection sort i.e. $\mathcal{O}(N^2)$ time ☹

Next class and next week..

- Wrap up the discussion on sorting
- Hashing: a very efficient method for search
- File handling in C
- Solving numerical problems using programming
- Future directions and wrapping up the course